MAPPING on UPMEM

Dominique Lavenier, Charles Deltel, David Furodet, Jean-François Roy

MAPPING on UPMEM

Dominique Lavenier, Charles Deltel,
David Furodet, Jean-François Roy

RESEARCH
REPORT

N° 8923

June 2016

Project-Team GenScale

ISSN 0249-6399

MAPPING on UPMEM

Dominique Lavenier[1], Charles Deltel[1]
David Furodet[2] , Jean François Roy[2]

Project-Team GenScale

Research Report N° 8923 — June 2016 —17 pages.

Abstract: This paper presents the implementation of a mapping algorithm on the UPMEM architecture. The mapping is a basic bioinformatics application that consists in finding the best location of millions of short DNA sequences on a full genome. The mapping can be constrained by a maximum number of differences between the DNA sequence and the region of the genome where a high similarity has been found. UPMEM's Processing-In-Memory (PIM) solution consist of adding processing units into the DRAM, to minimize data access time and maximize bandwidth, in order to drastically accelerate data-consuming algorithms. A 16 GBytes UPMEM-DIMM module comes then with 256 UPMEM DRAM Processing Units (named DPU). The mapping algorithm implemented on the UPMEM architecture dispatches a huge index across the DPU memories. DNA sequences are assigned to a specific DPU according to k-mers features, allowing to massively map in parallel million of them. Experimentation on Human genome dataset shows that speed-up of 25 can be obtained with PIM, compared to fast mapping software such as BWA, Bowtie2 or NextGenMap running 16 Intel threads. Experimentation also highlight that data transfer from storage device limits the performances of the implementation. The use of SSD drives can boost the speed-up to 80.

Key-words: Mapping, genomic banks, Processing-in-Memory

[1] INRIA / IRISA – dominique.lavenier@inria.fr

[2] UPMEM SAS – jroy@upmem.com

RESEARCH CENTRE
BRETAGNE ATALANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex France

MAPPING on UPMEM

Résumé : Ce rapport présente l'implémentation d'un algorithme de mapping sur l'architecture UPMEM. Le mapping est un traitement bioinformatique de base qui consiste à localiser sur un génome des millions de courtes séquences d'ADN. Le mapping peut être contraint par un nombre maximum de différences entre la séquence d'ADN et la région du génome où une forte similarité a été trouvée. La solution de *Processing-in-Memory* (PIM) UPMEM consiste à ajouter des unités de calcul dans la DRAM, pour minimiser le temps d'accès aux données et maximiser la bande passante, de manière à accélérer significativement les algorithmes gourmands en données. Une barrette UPMEM de 16G Mo propose donc 256 DRAM *Processing Units* (appelé DPU). L'algorithme de mapping implémenté sur l'architecture UPMEM réparti un index de grande taille sur l'ensemble des DPUs. Les séquences d'ADN à mapper sont aiguillées vers les DPUs en fonction de leur composition en k-mers, permettant ainsi une parallélisation massive du traitement. Les expérimentations effectuées sur des jeux de données du génome humain montrent un facteur d'accélération de 25 par rapport au logiciel BWA, Bowtie2 ou GenNextMap sur un serveur équipé de 16 cœurs Intel. Les expérimentations mettent aussi en avant que les entrées/sorties avec les organes de stockage limitent cette implémentation. L'usage de disques SSD permet de porter le facteur d'accélération à 80.

Mots clés : Mapping, banques génomique, Processing-in-Memory

1. Mapping

The mapping process consists in aligning short fragments of DNA, coming from high throughput sequencing, to a reference genome. Contrary to BLAST-like alignments that locate any portion of similar subsequences, the mapping action performs a complete alignment of the DNA fragments on the genome, by adding constraints such as the maximum numbers of substitutions or insertion/deletion errors.

Mapping software are intensively used in many bioinformatics pipelines as they represent basic treatments of sequencing data. The mapping aims to primarily detect small variations between a NGS dataset and a reference genome. This has multiple applications such as finding gene mutations in cancer disease, locating SNPs along the chromosomes of different varieties of plants, assembly genomes of very closed species, etc.

From a computer science point of view, the challenge is to be able to rapidly map hundreds of millions of short DNA fragments (from 100 to 250 bp) to full genomes, such as the Human genome (3.2×10^9 bp). The output of a mapping is a list of coordinates, for each DNA fragments, where matches have been found. As genome structures are highly repetitive, a DNA fragments can match at many locations. A mapping quality is thus associated. The quality value depends of the mapping confidence on a specific region. The output is generally encoded using SAM/BAM format [1]. It is a TAB-delimited text format consisting of a header section, which is optional, and an alignment section. Each alignment line has 11 mandatory fields for essential alignment information such as mapping position, and variable number of optional fields for flexible or aligner specific information.

Many mappers are available [3] [4]. They have their own advantages and drawbacks depending of several criteria such as speed, memory footprint, multithreading implementation, sensitivity or precision. The BWA mapper [2], based on Burrows-Wheeler Transform, can be considered as a good reference since it is often used in many bioinformatics pipelines. Examples of other mappers are Bowtie [5], NextGenMap [10], SOAP2 [6], BFAST [7] or GASSST [8].

2. UPMEM Architecture Overview

UPMEM technology is based on the concept of Processing-in-Memory (PIM). The basic idea is to add processing elements next to the data, i.e. in the DRAM, to maximize the bandwidth and minimize the latency. Host processor is acting as an orchestrator: It performs read/write operations to the memories and controls the co-processors embedded in the DRAM. This data-centric model of distributed processing is optimal for data-consuming algorithms.

A 16 GBytes UPMEM DIMM module comes with 256 processors: One processor every 64 MBytes of DRAM. Each processor can run its own independent program. In addition, to hide memory latency, these processors are highly multithreaded (up to 24 threads can be run simultaneously) in such a way that the context is switched at every clock cycle between threads.

The UPMEM DPU is a triadic RISC processor with 24 32-bits registers per thread. In addition to memory instructions, it comes with built-in atomic instructions and conditional branching bundled with arithmetic and logic operations.

From a programing point of view, two different programs must be specified: (1) the host program that will dispatch the data to co-processors memory, sends commands, input data, and retrieve the results; (2) the program that will specify the treatment on the data stored in the DPU memory. This is often a short program performing basic operations on the data. It is called a *tasklet*. Note however, that the architecture

of the UPMEM DPU allows different tasklets to be specified and run concurrently on different blocks of data.

Depending on the server configuration (i. e. the number of 16 GBytes UPMEM modules), a large number of DPU can process data in parallel. Each DPU only accesses 64 MBytes and cannot directly communicate with its neighbors. Data exchanges, if needed, must go through the host processor. A DPU has a fast working memory (64 Kbytes) acting as cache/scratchpad memory and shared by all tasklets (threads) running on the same DPU. This memory working space can be used to transfer blocks of data from the DRAM, and can be explicitly managed by the programmer.

To sum up, programing an application consists in writing a main program (run on the host processor) and one or several tasklets that will be executed on the DPUs. The main program has to synchronize the data transfer to/from the DPUs, as well as the tasklet execution. Note that the tasklet execution can be run asynchronously with the host program, allowing host tasks to be overlapped with DPU tasks.

3. Mapping implementation on UPMEM

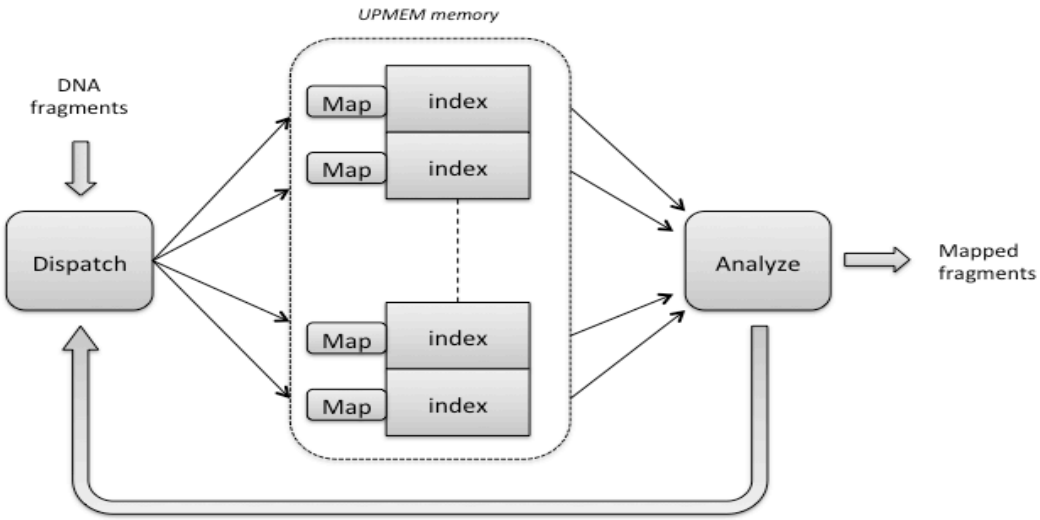
3.1 Overview

This section presents the mapping strategy elaborated to fully exploit the UPMEM architecture. The main idea is to distribute an indexing structure (computed from the genome) across the DPU memories. The host processor receives the DNA sequences and, according to specific features, dispatch them them to the DPUs. To globally optimize the treatment, group of DNA sequences are sent to the DPUs before starting the mapping process. Results are sent back to the host processor. DNA sequences that have not been mapped are reallocated to other DPUs for further investigation. A three pass processing allows more than 99% of DNA sequences to be mapped. This strategy supposes first to have downloaded the complete index into the DPU memory.

The following algorithm illustrates how the overall mapping process:

```
1:  Distribute the genome index across the DPUs
2:  Loop N
3:    List L_IN ← P DNA sequences
4:    Loop 3
5:        Dispatch sequences of list L_IN into DPUs
6:        Run mapping process
7:        Get results → 2 lists: L_GOOD & L_BAD
8:        Output L_GOOD
9:        L_IN ← L_BAD
```

The first loop (line 2) performs N iterations. N is the ratio of the number of DNA sequences to map divided by the number of DNA sequences that is processed in a single iteration. Typically, an iteration processes 10^6 sequences. The second loop (line 4) dispatches the sequences of the list L_{IN} into the DPUs. In the first iteration, the list L_{IN} contains all the DNA sequences. The mapping (line 6) is run in parallel and provides a mapping score (and coordinates) for all DNA sequences. The results are split into two lists (line 7): a list of sequences with good scores (list L_{GOOD}) and a list with bad scores (list L_{BAD}). Based on new criteria, the list L_{BAD} is dispatched to the DPUs in the 2nd and 3rd iterations. Another way to express the mapping is given by the figure below:

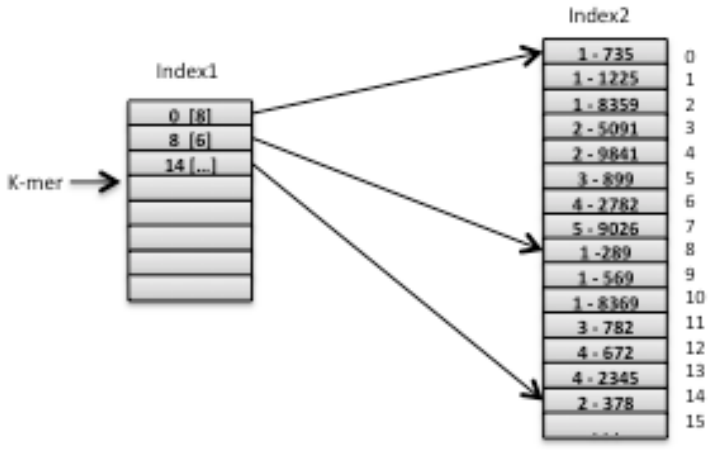


The UPMEM memory contains the genome index. The index is split across all DPUs. The host dispatches blocks of DNA fragments into the DPUs. Each DPU receives an average of N/P fragments (N = size of block, P = number of DPUs). DPUs send back to the host the mapping scores of the DNA fragments. If score are lower than a threshold value, a second mapping run is performed by reallocating the DNA fragments into other DPUs. Generally, after 3 iterations, DNA fragments are correctly mapped.

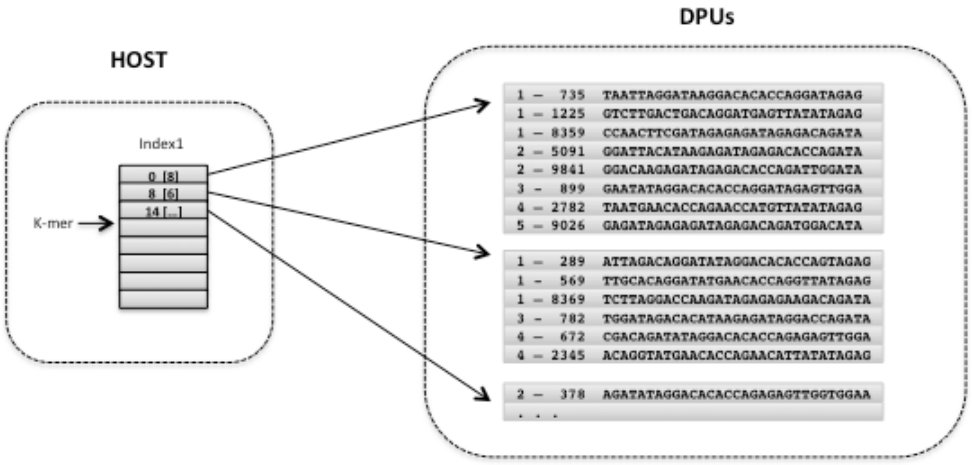
3.2 Genome Indexing

To speed up the mapping and to avoid to systematically comparing the DNA sequences with the full text of the genome, the genome is indexed using words of k characters, called k-mers. For each k-mers a list of coordinates specifying its location is attached, typically the chromosome number and the position of the k-mer on that chromosome. Then, the mapping process consists in extracting one or several k-mers from the DNA sequences in order to rapidly locate its position on the genome. The k-mer acts more or less as an anchor from which a complete match can be computed.

The index is composed of a first table of 4^K entries (Index1) that provides for all possible k-mer a list of coordinates where it occurs in the genome. The list of coordinates is stored in a second table (Index2). More specifically, for a specific k-mer, Index1 gives its address in Index2 and its number of occurrences. A line in Index2 indicates the chromosome number and a position on that chromosome.



The UPMEM implementation split Index2 into N parts, N being the number of available DPUs. As each DPU as a limited memory (64 MBytes), it cannot store the complete genome. Consequently, k-mer positions along the genome are useless inside a DPU. Thus, in addition to coordinates, portions of genome text corresponding to the neighborhood of the k-mers are memorized. The global indexing scheme is shown below.



Thus, for one k-mer, a line of Index2 memorizes the chromosome number (1 Bytes), the position of the k-mer on the chromosome (4 Bytes) and a neighborhood of 180 bp where each nucleotide is 2-bit encoded (45 Bytes). The storage one 1 k-mer requires 48 Bytes. Inside a DPU, 48 MBytes are allocated for the storage of the index or, in other words, the capability to store an equivalent genome of 1 Mbp. The rest of the memory is used for DNA sequences and result transfers.

3.3 Mapping algorithm

The host processor receives a flow of DNA sequences. For each sequence, a k-mer corresponding to the k first characters is extracted. Based on this k-mer, the DNA sequence is dispatched to the corresponding DPU. Every P sequences ($P = 10^6$), the host activates the DPUs to start the mapping process of the DNA sequences stored in each DPU.

More precisely, a specific DPU received an average of $Q = P/N$ DNA sequences. The mapping consists in comparing these Q sequences with the portions of the genome text stored inside each DPU memory, knowing that the k first characters are identical. The comparison algorithm can be more or less complex depending of the required mapping quality. For stringent mapping allowing only substitution errors, a simple Hamming distance can be computed. For mapping with indel errors, banded smith and Waterman algorithm can be performed.

However, this strategy doesn't guaranty to find all mapping locations. If an assembly error occurs along the k first characters, the DNA sequence will be dispatched to the wrong DPU and no correct mapping will be detected. Thus, for DNA sequences with a low score, the next k characters are taken into consideration to form a new k-mer allowing a new dispatching. If again, no good score are computed the next k characters are considered. Practically, after 3 iterations, the best matches are systematically found.

3.4 post processing

As the mapping is fully performed inside the DPUs, no more computation is required. The post processing consists simply in getting the results from the DPUs and formatting the data before writing them to disk.

4. Performance evaluation

Performances have been evaluated on the Human genome with a DELL server (Xeon Processor E5-2670, 40 cores 2.5 GHz, 64 GBytes RAM) configuration running Linux Fedora 20. As I/O transfer has a great impact on the performances, we measure the hard disk read speed of the server using the **dd** Linux command.

```
ll refseq_prot.fasta
-rwxrwxrwx 1 root 24101720035 Mar  3 21:07 refseq_prot.fasta*
dd if=refseq_prot.fasta of=/dev/null
47073671+1 records in
47073671+1 records out
24101720035 bytes (24 GB) copied, 184.908 s, 130 MB/s
```

On this server, the bandwidth for reading data from the **local disk** is equal to 130 MB/sec

The transfer times between the UPMEM memory and the host CPU are taken from a rigorous performance evaluation performed on a Intel core i7 6700 (SKYLAKE) at 3.4 GHz with a 2xDDR4 channel [11]:

Data:

- Human genome: 3.2 G bp
- DNA sequences: a set of 111 x 10^6 100 bp sequences (13 GBytes)

To store the index corresponding to the Human genome, the minimum number of DPUs is equal to $3.2x10^9/10^6$ = 3200 DPUs (cf section 3.2: a DPU store the equivalent of 1Mbp). The UPMEM configuration is thus set to 3328 DPUs (13 DIMM modules).

We evaluate the execution time according to the algorithm of section 3:

1. Distribution of the genome index across the DPUs
2. Loop:
   2.1 Dispatching of the sequence to the DPUs
   2.2 Mapping
   2.3 Result analysis

Distribution of the genome index across the DPUs: **30 sec**

This step can be divided into the two following actions

- Download the index from the storage device
- Dispatch the index inside the DPUs

As most of the mappers, the index is pre-computed. In out case, Index1 is fully pre-computed, and the genome is formatted to facilitate its encoding into the DPU memories. The size of Index1 is determined by the k-mer length. Here, the k-mer length is set to 13. The numbers of entries of Index1 is thus equal to 4^{13} = 64 M entries. One entry stores the number of k-mers (1 integer) and an address in index2 (1 integer). Thus the total size of Index1 is 256 MBytes. The size of the file containing the formatted genome is equal to the size of the genome (3.2 GBytes). The full information to download represents approximately 3.5 GBytes. The download time is constrained by the I/O bandwidth of the local disk. With a bandwidth of 130 MB/sec the download time is equal to 27 sec.

Dispatching Index2 across the DPUs consists in writing for each k-mers of the genome 48 bytes in a DPU memory, that is globally $3.2 \times 10^9 \times 48 = 153.6$ GBytes.

The bandwidth for transferring data form the Host memory to the DPU memories is estimated to 11.53 GB/s (see [11]). The time for transferring the index is thus equal to 153.6/11.53 = 13.3 sec. With the associated overhead to format the data, we globally estimate this step to 15 sec.

Actually, these two actions (download and dispatch) can be easily overlapped, leading to a global initializing phase estimated to 30 sec.

Loop: **1 sec.**

The loop performs the following actions:

1. Get block of fragments from disk.
2. Dispatch the fragments to the DPUs
3. Initialize and start the DPUs
4. Perform the mapping
5. Get Results from DPU
6. Analyze and write results

1. Get block of 106 fragments from disk: 1sec

In our implementation, a loop iteration processes 10^6 DNA sequences. These sequences are read from the storage device. One million sequences of length 100 in Fasta format represent approximately 130 MBytes of data (text sequence + annotation). The time to read this information depends again of the I/O bandwidth of the storage device. With a bandwidth of 130 MB/sec, the time is equal to 1 sec.

2. Dispatch the fragments to the DPUs: 50 ms

Dispatching the DNA sequences to the DPUs is fast: it consists in coding the 13 first characters of the sequence and in copying the sequence to the target DPU. Experiments indicates an execution time < 40 ms. Transferring 100 MBytes of data to the DPU memory is also very fast. It requires 0.1/11.5 = 8.7 ms. Overall, this step takes a maximum of 50 ms.

3. Initialize and start the DPUs: 23 µs.

A DPU run 10 tasklets. Each tasklet receives two parameters: the number of DNA fragments to process, and the address where these fragments are stored. This represents 2 integers (8 bytes) by tasklet, or 80 bytes per DPU, or an overall transfer of 80x3328 = 266240 bytes. The equivalent time is: $266240/11.53 \times 10^9 = 23$ µs. As broadcasting commands to 128 DPU simultaneously is possible, booting the DPU consist in sending 3328/128 = 26 commands. This time is negligible.

4. Mapping: 40 ms

In average, a DPU receive $10^6/3328 = 300$ DNA sequences to process. The number of occurrences of a k-mer of size 13 is approximately the size of the genome divided by 4^{13}, that is $3.2 \times 10^9/4^{13} = 50$. The number of mappings that must be executed by one DPU is thus equal to 15000. The simulations executed on the UPMEM Cycle Accurate Simulator range from 10×10^6 to 25×10^6 cycles to perform such a treatment, depending of the DPU load. As a matter of fact, the repartition inside the DPUs is generally not uniform. It depends of the nature of the DNA sequences. We have to take into account the worst execution time since all DPUs must finish before analyzing all results.

In the second and third round, only a fraction of the DNA sequences that have not matched are sent to other DPUs. It represents less than 10% of the initial number of sequences. The impact on the overall execution time is weak. An upper bound estimation for the 3 loop iteration is 30×10^6 cycles, leading to an execution time of 40 ms with a 750 MHz frequency.

5. Get results: 0.7ms

For each DNA fragments, the DPU output the following information: genome coordinates and mapping scores (2 integers). There are thus 2 x 4 x 10^6 = 8 M bytes to transfer. The transferring time = 0.7ms

6. Analysis & write results: 100 ms

This step evaluates the score of the mapping and selects DNA sequences that have to be analyzed again. It also writes results to the output file. Our experimentation estimates the execution time to 0.1 sec in the worst case.

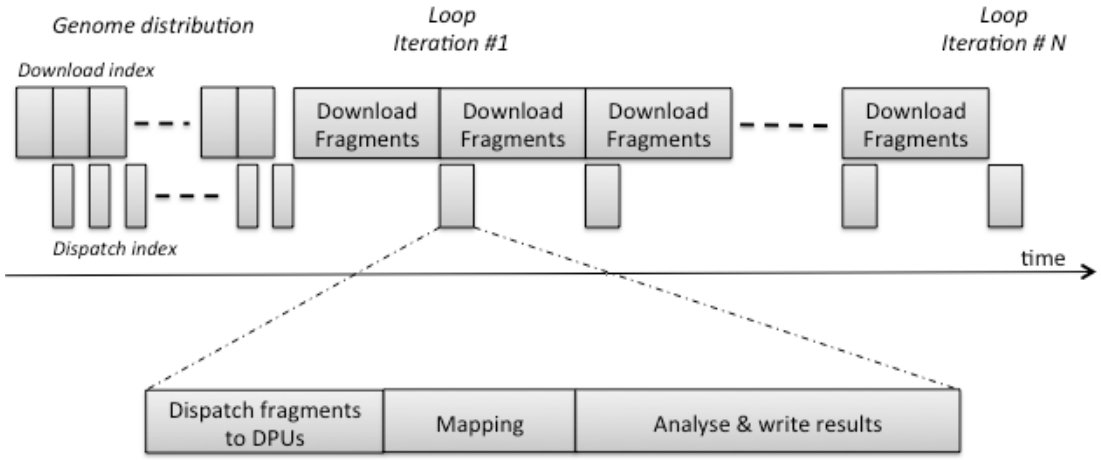
Actions 2 to 6 are iterated 3 times. The first time involves all DNA fragments, the second time less than 10% and the third time less than 3%. The cumulated execution time of actions 2 to 6 is thus approximately equal to:

1.13 x (50 + 40 + 100) = 190 ms.

Actually, getting the data from the disk (action 1) can be overlapped with the other tasks (actions 2 to 6), leading to an iteration execution time of 1 second.

Overall Execution Time

Put together, with I/O overlapping, the overall scheduling of the different tasks can be represented as shown below:



We can approximate the execution time by the time for uploading the genome index to the UPMEM memory (30 sec) plus the time to perform N loop iterations (N x 1 sec). For the data set we used (111 x 10^6 fragments), the overall execution time is:

Overall execution time: 30 + 111 = 141 sec.

As the I/O bandwidth has a great impact on the overall performances, we test the implementation with a 512 GB SSD drive present on the server. We measure an average bandwidth of 700 MB/sec.

In such a case, the genome distribution execution time is now dominated by the dispatch index execution time (15 sec). For the loop execution time a good balance is achieved between the time for getting the fragments (185 ms) and the time for executing actions 2 to 6 (190 ms). Taking a safe execution time of 0.2 second per iteration, the new overall execution time is

Overall execution time with SSD: 15 + 111 x 0.2 = 37.2 sec.

5. Comparison with other mappers

Comparison have been made with the following mappers:

- BWA [2]
- Bowtie2 [5]
- NextGenMap [10]

The three software have been run with different number of threads: 8, 16 and 32 (commands are given in annex 1).

	8 threads	16 threads	32 threads
BWA	5901	3475	2191
Bowtie2	5215	2916	2241
NextGenMap	3485	2104	1552

Time is given in second

Speed-up

The speedup is calculated as the ratio between the reference software execution time and the estimated UPMEM execution time. It considers both a hard disk and a SSD disk.

	8 threads		16 threads		32 threads	
	Hard	SSD	Hard	SSD	Hard	SSD
BWA	41	157	24	93	15	58
Bowtie2	36	140	20	78	16	60
NextGenMap	24	93	15	56	11	41

Cost Optimized *Performance Optimized*

For cost optimized (Hard disk, 8 threads) configuration, the speed up brought by Processing-In-Memory is 41 times for BWA, 36 for Bowtie2 and 24 for NextGenMap. For performance optimized (SSD, 32 threads) configuration, the speed up brought by Processing-In-Memory is 58 for BWA, 60 for Bowtie2 and 41 for NextGenMap.

Using BWA, Bowtie2 or NextGenMap software with a SSD drive has a negligible impact on the overall performances (no significant improvement).

Quality Evaluation

The DNA sequence dataset is a synthetic dataset generated from the reference human genome. Only substitution errors have been considered. This is the most frequent error of the Illumina technology. In that case, the mapping consists in computing a Hamming distance between the DNA sequence and a region of the genome of identical size. BWA, Bowtie2, NextGenMap or UPMEM results are nearly identical. For

some specific cases where errors are concentrated at the beginning of the sequence, we miss a few mapping. On the other hand, other software do not report a few mapping when the number of errors is too large.

To address indel errors, the solution would be to implement a banded smith and waterman algorithm, which is much more time consuming. But running this algorithm, knowing that more than 95% of the errors don't fit in this category, is a costly solution. In our case, this algorithm could be only fired on rounds 2 & 3. The impact on the mapping execution time will be then very limited.

Bibliography

1. The SAM/BAM Format Specification Working Group, Sequence Alignment/Map Format Specification, April 2015, https://samtools.github.io/hts-specs/SAMv1.pdf
2. Li H. and Durbin R. (2009) Fast and accurate short read alignment with Burrows-Wheeler Transform. Bioinformatics, 25:1754-60.
3. Jing Shang, Fei Zhu, Wanwipa Vongsangnak, Yifei Tang, Wenyu Zhang, and Bairong Shen, Evaluation and Comparison of Multiple Aligners for Next-Generation Sequencing Data Analysis, BioMed Research International, vol. 2014, Article ID 309650, 16 pages, 2014.
4. Schbath S, Martin V, Zytnicki M, Fayolle J, Loux V, Gibrat J-F. Mapping Reads on a Genomic Sequence: An Algorithmic Overview and a Practical Comparative Analysis. Journal of Computational Biology. 2012;19(6):796-813. doi:10.1089/cmb.2012.0022.
5. Langmead B. Trapnell C. Pop M., et al. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. 2009;10:R25.
6. Li R. Yu C. Li Y., et al. SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics. 2009;25:1966–1967
7. Homer N. Merriman B. Nelson SF. BFAST: an alignment tool for large scale genome resequencing. PLoS ONE. 2009;4:e7767
8. Rizk G. Lavenier D. GASSST: global alignment short sequence search tool. Bioinformatics. 2010;26:2534–2540
9. Ayat Hatem, Doruk Bozdağ, Amanda E Toland, Ümit V Çatalyürek, Benchmarking short sequence mapping tools, BMC Bioinformatics 2013, 14:184
10. Fritz J. Sedlazeck, Philipp Rescheneder, and Arndt von Haeseler NextGenMap: fast and accurate read mapping in highly polymorphic genomes. Bioinformatics (2013) 29 (21): 2790-2791 first published online August 23, 2013 doi:10.1093/bioinformatics/btt468
11. UPMEM DPU-Data exchange with main CPU. UPMEM Technical note, version 1.3

Annex 1: MAPPER execution time

Data:

- Human genome: 3.2 G bp
- DNA sequences: a set of 111 x 10^6 100 bp sequences (13 GBytes)

BWA: version 0.7.12-r1039

32 threads

```
time bwa mem -t 32 HG38 HG38read.fa > outfile
67467.819u 662.619s 36:31.84 3108.3% 0+0k 10803480+8io 41pf+0w
```

16 threads

```
time bwa mem -t 16 HG38 HG38read.fa > outfile
54002.790u 888.145s 57:55.43 1579.3% 0+0k 10563392+0io 12pf+0w
```

8 threads

```
time bwa mem -t 8 HG38 HG38read.fa > outfile
46519.108u 447.863s 1:38:21.97 795.7% 0+0k 10802040+0io 4pf+0w
```

NextGenMap; version 0.5.0

32 threads

```
time ngm -q HG38read.fa  -r HG38.fasta -o outfile -t 32
40051.515u 626.981s 25:52.84 2619.6% 0+0k 18112152+48io 30pf+0w
```

16 threads

```
time ngm -q HG38read.fa  -r HG38.fasta -o outfile -t 16
29655.900u 412.679s 35:04.03 1429.0% 0+0k 19110920+72io 1pf+0w
```

8 threads

```
time ngm -q HG38read.fa  -r HG38.fasta -o outfile -t 8
25983.340u 286.026s 58:05.82 753.6%  0+0k 19773048+8io 26pf+0w
```

Bowtie2: version 2.2.9

32 threads

```
time $BT2_HOME/bowtie2 -x HG38 -f HG38read.fa -S toto -p 32
66058.337u 4786.631s 37:21.32 3160.8% 0+0k 19632568+0io 13pf+0w
```

16 threads

```
time $BT2_HOME/bowtie2 -x HG38 -f HG38read.fa -S toto -p 16
43463.622u 2564.357s 48:36.19 1578.3% 0+0k 19815728+0io 22pf+0w
```

8 threads

```
time $BT2_HOME/bowtie2 -x HG38 -f HG38read.fa -S toto -p 8
40011.927u 1231.559s 1:26:55.78 790.7% 0+0k 20104872+0io 16pf+0w
```

Annex 2: Tasklet code

```
void initTX(int8_t *TX) {
  int i,k;
  for (i=0; i<256; i++) {
      k=0;
      if ((i&3) != 0)k++;
      if (((i>>2)&3)!=0) k++;
      if (((i>>4)&3)!=0) k++;
      if (((i>>6)&3)!=0) k++;
      TX[i] = k;
    }
}

int getCKEY(char *seq, int sz) {
  int i,c;
  int key = 0;
  for (i=0; i<sz; i++) {
      c = codeNT(seq[i]);
      key = (key<<2)+(c&3);
    }
  return key;
}

int TN()
{
  int x, k, i, n, nr, xr, adr, lread, nb_pos, shift, adr_pos, score, min_score, min_seq;
  int    *RES    = dma_alloc(32);
  int    *INFO   = dma_alloc(32);
  char   *READ   = dma_alloc(128);
  int8_t *GENOME = dma_alloc(32);
  int8_t *TX     = dma_alloc(256);
  int8_t *RC     = dma_alloc(32);
  int NB_READ  = ((int*) mbox_recv())[0]; // get values from the private mailbox
  int ST_READ  = ((int*) mbox_recv())[1];
  initTX(TX);
  for (nr=ST_READ; nr<NB_READ; nr=nr+NB_TASKLET) {
      adr = ADDR_INFO + (nr<<5);  // get information on the read to process
      mram_ll_read32(adr,INFO);
      adr_pos = INFO[0];  nb_pos = INFO[1];
      lread   = INFO[2];  shift  = INFO[3];
      adr = ADDR_READ + (nr<<7); // get the read
      mram_ll_read128(adr,READ);
      for (i=0; i<32; i++) RC[i] = getCKEY(&READ[i<<2],4);
      min_score = 1000;
      for (n=0; n<nb_pos; n++) {
         adr = (adr_pos+n)<<5;  // get part of the genome
         mram_ll_read32(adr,GENOME);
         score = 0;
         for (i=0; i<lread>>2; i++) {
             k = (int) (RC[i]^GENOME[i+shift]);
             k = k&0xFF;
             score = score + TX[k];
             if (score > min_score) break;
           }
         if (score < min_score) { min_score = score; min_seq = n; }
       }
      RES[0] = min_score + (min_seq<<16);
      adr = ADDR_RESULT + (nr<<5);
      mram_ll_write32(RES,adr);
    }

  return 0;
}
```

RESEARCH CENTRE
BRETAGNE ATALANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex France

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr
ISSN 0249-6399